5 Hashing

5.1 Introduction

Reading: MAW 5.1 and 5.2

- Motivation: Insertion, deletion, and search.

Worst-case	I	D	S
array	$O(1)$	$O(n)$	$O(n)$
sorted array	$O(n)$	$O(n)$	$O(\log n)$
pointer	$O(1)$	$O(n)$	$O(n)$
BST	$O(n)$	$O(n)$	$O(n)$
AVL	$O(\log n)$	$O(\log n)$	$O(\log n)$
B tree	$O(\log n)$	$O(\log n)$	$O(\log n)$

  Question: Can we design a data structure such that search can be done in constant time?

- Idea: Given a key, if we know its address (index) in the array, then search can be done in $O(1)$ by direct access.

  Can we define the address of a key as a function of the key? This is the idea of hashing.

- Hashing:

  Hash table HT: An array of b buckets (or cells or addresses) indexed from 0 to $b-1$.

  Hash function h: $\{keys\} \to \{0,1,\ldots,b-1\}$. Key x is therefore stored in address $h(x)$ (or $HT[h(x)] = x$).

- Example: A company of 100 employees. The access key is a nine-digit number (SSN). The array (hash table) is indexed from 0 to 99. How can we define the hashing function h?

  - $h(d_1d_2d_3d_4d_5d_6d_7d_8d_9) = d_4d_5$.
  - $h(d_1d_2d_3d_4d_5d_6d_7d_8d_9) = d_1 \cdots d_9 \bmod 100$.
  - $h(d_1d_2d_3d_4d_5d_6d_7d_8d_9) = (\sum_{i=1}^{9} d_i) \bmod 100$.

- Commonly used hash functions:

  A good hash function is such that each key is equally likely to be hashed to any of the b addresses in the hash table (so that you don't over-use any address). This goal is almost impossible to achieve. The followings are some commonly used hash functions:

  - The division method: $h(x) = x \bmod b$.
  - The multiplication method: $h(x) = \lfloor b(xA - \lfloor xA \rfloor) \rfloor$, for $0 < A < 1$.
    By Knuth, a good choice for A is the negative conjugate of the golden ratio, $(\sqrt{5}-1)/2 = 0.618....$

  What if the keys are not integers?

  Ways to convert a character string, *pt* $(112, 116)$, into an integer:

  - Sum of the ASCII numbers: $112 + 116 = 228$:
  - Weighted sum of the ASCII numbers: $1 \times 112 + 2 \times 116 = 344$; and
  - Radix-128 integer: $112 \times 128 + 116 = 14452$.

- Collision:

  Ideally, we want h to be easy to compute and $h(x) \neq h(y)$ for $x \neq y$.

  But the number of possible keys is often much larger than b, the size of the hash table. Why?

  Consider the example of the 100 person company. The number of possible keys is $10^9 = 1$ billion, and the number of keys in use is 100. If we create a hash table with size 1 billion, 99.99999% of space will be empty. If we create

a hash table of size 100, since we have no idea how the keys in use are distributed, it is almost impossible to avoid having two keys mapped to the same address in the hash table.

Collision happens if $h(x) = h(y)$ for $x \neq y$.

Collision is inevitable.

What can we do once collision occurs?

5.2 Separate chaining (Open hashing)

Reading: MAW 5.3

- The hash table is an array of pointers. $HT[i]$ points to a linked list of keys x such that $h(x) = i$.
- Time complexity:

  Let n be the number of keys in the hash table currently.

  - Worst-case length of a linked list is n. Worst-case time occurs when the entire linked list is traversed.
  - Average-case length of a linked list is $\frac{n}{b}$. Average-case time occurs when half the linked list is traversed.

Worst-case time	I	D	S
Worst-case length	1	n	n
Average-case length	1	$\frac{n}{b}$	$\frac{n}{b}$

Average-case time	I	D	S
Worst-case length	1	$\frac{n}{2}$	$\frac{n}{2}$
Average-case length	1	$\frac{n}{2b}$	$\frac{n}{2b}$

5.3 Open addressing (Closed hashing)

Reading: MAW 5.4 and 5.5

- Use an array for the hash table ($b \geq n$), with the keys stored in the array directly.

  To insert x, if $HT[h(x)]$ is already occupied by another key (i.e., a collision is detected), alternate cells are tried until an empty cell is found.

  Probing sequence: $h_0(x), h_1(x), h_2(x), h_3(x), \ldots$, where $h_i(x) = (h(x) + f(i)) \bmod b$ and $f(0) = 0$. Function f decides the size of the jumping step to reach the next cell. Note that $h_0(x) = h(x)$.

  For example, $f(0) = 0$, $f(1) = 1$, $f(2) = 3$, $f(3) = 7$, etc.. Show the cells being tried.

- Linear probing: Choose a linear function as f, e.g., $f(i) = i$, in which case the probing sequence is $h(x), h(x) + 1, h(x) + 2, \ldots$.

  Primary clustering: Keys have the tendency to be stored next to each other to form blocks, increasing the time complexity for future insertion.

  For example, assume there are $n = \frac{b}{2}$ keys in the table currently, occupying all even-indexed cells, $0, 2, 4, \ldots$. An unsuccessful search takes $\frac{1+2}{2} = 1.5$ probes on average. However, if these $n = \frac{b}{2}$ keys occupy the first half of the table, $0, 1, 2, \ldots$, an unsuccessful search takes $(((\frac{b}{2} + 1) + (\frac{b}{2}) + \cdots + 2) + \frac{b}{2})/b = \frac{b}{8} + \frac{5}{4}$ probes on average.

- Quadratic probing: To eliminate primary clustering, choose a quadratic function as f, e.g., $f(i) = i^2$, in which case the probing sequence is $h(x), h(x) + 1, h(x) + 4, h(x) + 9, \ldots$.

  In linear probing, as long as there is an empty cell, a key can always be inserted. However, this is not the case with quadratic probing.

  For example, let $b = 4$ and assume that cells 0 and 1 are occupied. We wish to insert x with $h(x) = 0$. Quadratic probing with $f(i) = i^2$ will only probe cells 0 and 1, thus x can never be inserted.

Theorem: For quadratic probing, if b is prime and the hash table is at least half empty, successful insertion is guaranteed.

Secondary clustering: For $x \neq y$, if $h(x) = h(y)$, they have the same probing sequence.

- Double hashing: To eliminate both primary and secondary clustering, two hash functions, h and h', are used. Define $h_i(x) = (h(x) + ih'(x)) \bmod b$. (This is the same as defining $f(i) = ih'(x)$.) The probing sequence is $h(x), h(x) + h'(x), h(x) + 2h'(x), \ldots$.

  The distance between two adjacent probings is $h'(x)$, determined by x. Unless collision occurs for both functions at x and y, i.e., $h(x) = h(y)$ and $h'(x) = h'(y)$, two different keys won't have the same probing sequence.

- Rehashing: We can also try to reduce the chance of collision by keeping the hash table less full. When the hash table has a lot of items, we double the size of the hash table by creating a new one about twice as big as the previous one. This process is called rehashing.

  There are three strategies commonly used to decide when to rehash.

  - Rehash as soon as the table is half full.
  - Rehash only when an insertion fails.
  - Rehash when the table reaches a certain load factor, which is the ratio of the number of items in the hash table to the size of the table.